Improving Reusability in Software Process Lines

Emmanuelle Rouillé, Olivier Barais, Benoit Combemale, Touzet David, Jean-Marc Jézéquel

Improving Reusability in Software Process Lines

Emmanuelle Rouillé*†, Benoît Combemale†, Olivier Barais†, David Touzet* and Jean-Marc Jézéquel†
*Sodifrance
P.A. la Bretèche, avenue Saint-Vincent, 35768, Saint-Grégoire, France
Email: {erouille, dtouzet}@sodifrance.fr
†Université de Rennes 1
IRISA, Campus de Beaulieu, 35042, Rennes, France
Email: {emmanuelle.rouille, benoit.combemale, olivier.barais, jean-marc.jezequel}@irisa.fr

***Abstract*—Software processes orchestrate manual or automatic tasks to create new software products that meet the requirements of specific projects. While most of the tasks are about inventiveness, modern developments also require recurrent, boring and time-consuming tasks (*e.g.,* the IDE configuration, or the continuous integration setup). Such tasks struggle to be automated due to their various execution contexts according to the requirements of specific projects. In this paper, we propose a methodology that benefits from an explicit modeling of a family of processes to identify the possible reuse of automated tasks in software processes. We illustrate our methodology on industrial projects in a software company. Our methodology promoted both the identification of possible automated tasks for configuring IDEs and continuous integration, and their reuse in various projects of the company. Our methodology contributes to the companies' efficiency, including their agility and ability to experiment new practices, while remaining focused on solving business problems.**

I. Introduction

When comes up the time to develop new software products, software companies face to orchestrate manual or automatic tasks in a given software process to meet specific requirements. While most of the tasks are about inventiveness, modern developments also require recurrent, boring and time-consuming tasks, occurring several times during a project or across different projects. For instance, configuring the development environment (*e.g.,* IDE, code versioning) occurs for each software developer of each project.

While automating such tasks would improve the productivity of a software company, implementing *automation components* to automate them is still an hard task. Indeed, the variability of the requirements across the various projects implies variability of the corresponding software processes, raising various possible contexts of use of the automation components. Therefore, the difficulty is to implement automation components that are reusable across their different contexts of use. For instance, an automation component in charge of putting some source code under version control should be reusable whatever the URL of a remote repository.

Of course, mechanisms exist that enable to implement reusable components (*e.g.,* parameterization, modularization...). But using these mechanisms first requires to have identified the parts of a component that vary and that do not vary. In other words, using these mechanisms requires the identification of the *level of reuse* of a component.

Some approaches [1]–[3] provide guidelines to identify the level of reuse of software components that realize software products, by relying on the specification of the variability of these software products. However, in this article we are interested in identifying the level of reuse of components that realize software processes (*i.e.,* automation components), and the guidelines provided by these approaches are not reusable in this case. The main reason is because these guidelines rely on the fact that each part of a software product is realized by a software component, whereas in the case of software processes each part of a software process is not necessarily automated by an automation component.

In this article we propose a methodology to address this problem and improve the capitalization of highly reusable automation components. Our methodology consists of binding the automation components to the work units they automate in a family of software processes. This binding enables the identification of the different contexts of use of the automation components, which is the necessary input to identify their level of reuse. We rely on Software Process Line Engineering (SPrLE) [4] in order to define the family of processes. We illustrate the application of our methodology on Java development projects of a software company, namely Sodifrance[1].

This paper is organized as follows. Section II presents the background of our work. It first presents the process modeling language SPEM 2.0 [5], that we use to give an overview of our methodology. It then gives an overview of SPrLE. We present our methodology in Section III and we illustrate its application on an example in Section IV.We discuss our methodology in Section V. The related work is discussed in Section VI. We conclude and present our perspectives of work in Section VII.

II. Background

This section presents the SPEM Software Process Modeling Language (SPML) and SPrLE.

A. SPEM 2.0

We only introduce the subpart of SPEM 2.0 required to understand the illustrations of the process of our methodology as well as the principle of the previous work we rely on. The SPEM 2.0 specification provides the concepts of task (),

[1]http://www.sodifrance.fr/

work product ( ), role ( ) and tool ( ). A task is a work to realize during the process execution. Tasks take zero or several work products as inputs and outputs. Zero or several roles perform a task. Zero or several tools support the realization of a task.

In order to model the flow of tasks, we use the concepts of control flow ([guard]), initial node ( ), final node ( ), decision node ( ), fork node ( ) and join node ( ) from UML 2 [6] activity diagrams.

B. SPrLE

SPrLE [4] relies on Software Product Line Engineering (SPLE) [7] to manage the variability of software processes. Indeed, SPLE consists in specifying commonalities and variabilities of software products in order to reuse commonalities. So does SPrLE, where software products are software processes. In SPrLE, an SPrL defines in intention (i.e., by factorizing the common parts between software processes) a set of software processes [8]. There exist several approaches that implement an SPrL [8]–[12]. These approaches rely on model-driven engineering to define the processes of the SPrL as well as their commonalities and variabilities.

III. Methodology

In this section we present our methodology for improving the identification of the level of reuse of the automation components (ACs). An AC is a software component that automates a manual recurrent task that occurs during software processes. Fig. 1 shows an overview of our methodology.

Our methodology involves a process expert (*i.e.,* someone who knows the different processes of a company) and an architect (*i.e.,* someone able to identify the manual recurrent tasks to automate and able to implement components to automate these tasks).

The process expert starts by defining an SPrL (step 1). To this aim, the process expert uses an SPML to define the processes of the SPrL. The process expert can use the same SPML to define the variability of the processes or can use another modeling language, according to the process expert's needs and to the mechanisms provided by the SPML.

The architect then identifies the ACs that are useful for the company (step 2). During this step the architect is not concerned with the identification of the level of reuse of the ACs. The architect is only concerned with the identification of the existing ACs and of the future ACs (by identifying the manual recurrent tasks that would benefit to be automated).

Then, the architect uses a binding modeler to model the binding between the ACs (identified in step 2) and the work units (*e.g.,* tasks) of the SPrL they contribute to automate (step 3). The work units binded to an AC represent its contexts of use. If several ACs contribute to the automation of a same work unit, the binding specifies in which order these ACs must execute. If a work unit varies, then the binding specifies which variant of this work unit an AC automates. We consider that a work unit varies either if this is this work unit itself that varies or if they are the model elements directly related to this work unit (*e.g.,* tools, roles, work products, control flows) that vary. Finally, the binding specifies if an AC contributes to the automation of the initialization, execution and/or finalization of a work unit. Specifying an initialization (resp. finalization) of a work unit prevents the omission of this initialization (resp. finalization). Indeed, the initialization (resp. finalization) is in this case tightly coupled to the work unit it initializes (resp. finalizes). This is not the case when the initialization (resp. finalization) is replaced by a work unit that occurs before (resp. after) the initialized (resp. finalized) work unit.

Then the architect implements the ACs (step 4). During this step, the binding model enables to improve the identification of the level of reuse of the ACs. Indeed, by thinking about how to implement an AC in order to cover all its contexts of use, the architect is able to determine the parts of the AC that vary. Then, the architect uses mechanisms (*e.g.,* parameterization, modularization...) to implement the reuse of the ACs. By implementing the ACs the architect may realize that the binding model is incorrect. For instance, the implementation of an AC may contain parts that are irrelevant for some of its contexts of use. In this case, the architect goes back to step 3 in order to correct the binding model.

IV. Illustration of the Methodology

In this section, we show an example of application of our methodology on a Shell script that automates the configuration of the local workspace of a developer. This script is used during Java development projects of the Sodifrance company. It is executed at the beginning of the development activity by each developer of the project. It is also executed each time a new developer integrates a Java development project whose development activity is already started.

Fig. 2 shows an extract of this Shell script. In this Fig. this Shell script is in the state it was before the application of our methodology, *i.e.,* configured to be used with a specific project. This Shell script takes as parameters (line 1) the path of the local workspace (*wsPath*), the path of the Maven local repository of the developer (*m2Path*) and the URLs of the repositories of the source code (*sourceUrl*) and of the build code (*buildUrl*). The build code corresponds to other resources useful to the workspace's configuration. The script automates the following steps:

1) SVN checkout of the source code (l. 10) and of the build code (l. 13),
2) Maven compilation of Java projects (l. 18),
3) Maven configuration of Eclipse projects (l. 19),
4) Maven configuration of the Eclipse workspace (l. 20),
5) Buckminster import of the projects into the Eclipse workspace (from l. 23).

We know describe the details of the SPrL capturing the family of Java development processes of the Sodifrance company that are useful to understand the following. This SPrL specifies that two Version Control Systems (VCS) can be used: either Git or SVN. Furthermore, on the shared repository that contains the build code, there are Buckminster properties that are stored in their own folder. The SPrL specifies that the

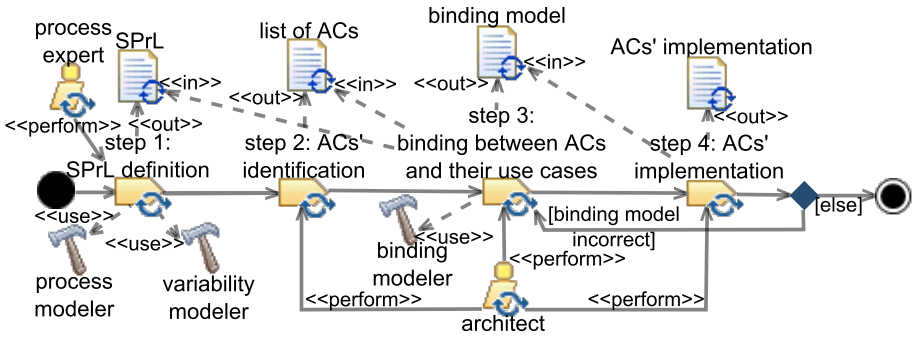

Figure 1: Methodology overview

```
1  Param([string] $wsPath, [string] $m2Path, [string]
       $sourceUrl, [string] $buildUrl)
2
3  # Variable settings
4  $sourcePath = $wsPath + "/source"
5  $buildPath = $wsPath + "/build"
6
7  [...]
8
9  # Checkout of source code from URL  $sourceUrl to
       folder $sourcePath
10 svn checkout $sourceUrl $sourcePath
11
12 # Checkout of build code from URL $buildUrl to folder
       $buildPath
13 svn checkout $buildUrl $buildPath
14
15 # Run Maven commands to compile and configure Eclipse
       projects and to configure the workspace
16 $sourcePom = $sourcePath + "/pom.xml"
17 [...]
18 mvn compile -f $sourcePom
19 mvn eclipse:eclipse -f $sourcePom
20 Invoke-Expression ("mvn -D eclipse.workspace=" +
       $wsPath + " eclipse:configure-workspace")
21
22 # Run Buckminster commands to import projects into the
       workspace
23 $buckyBuild = $buildPath + "/buckybuild.properties"
24 [...]
25 $cQueryFile=$buildPath + "/org.eclipse.buckminster.
       myproject.build/buckminster/myproject-build.cquery
       "
26 buckminster import --properties $buckyBuild -data
       $wsPath $cQueryFile
27 [...]
```

Figure 2: Extract of a Shell script example that configures an Eclipse workspace

path of this folder on the shared repository vary according to the projects. Finally, the Shell script uses a Buckminster component query to perform the step 5. The component query specifies what to import in the Eclipse workspace. The SPrL does not specify any variability about this component query. This means that it is common to all the projects.

We know show that the information of the SPrL is useful to improve the level of reuse of this script.

In the script the step 1 depends on SVN, while the steps 2 to 5 are independent of the VCS. Furthermore, the SPrL specifies that the VCS can change. Therefore, this motivates the need for decoupling the step 1 from the script in order to be able to reuse the other steps independently of this one.

On the other hand, the line 23 of the script assigns a variable corresponding to the path of the Buckminster properties. However, the SPrL specifies that the path of the properties, into the folder represented by the variable named *$buildPath*, could vary according to the projects. Therefore, this motivates the need for passing the second part of the assignment of the variable named *$buckyBuild* (*i.e., "/buckybuild.properties"*) as a parameter of the script.

Finally, the line 25 assigns a variable corresponding to the path of the Buckminster component query, which is generic according to the SPrL. However, the architect can observe that in the script the path of the component query is specific to the current project. Indeed, this path depends on the variable named *$buildPath*, which represents the path on the local environment of the developer where the build code has been checked out. And the component query has been copied in the local environment of the developer during the checking out of this build code. However, this build code has been checked out from a repository that is specific to the current project. This means that the component query is stored on this repository that is specific to the current project. This makes the reuse of the component query error prone and time consuming for other projects. Indeed, the component query would have to be copied on the repositories corresponding to these projects, resulting in duplications. Therefore, this motivates the need for removing from the script the dependence between the path of the component query and a specific project. A solution would be to store the component query on a repository independent of any project, to add in the current script a command that checks out this component query, and to assign the new path of the component query to the variable named *$cQueryFile*.

V. Discussion

In this section we discuss the benefits, the limitations and the disadvantages of our methodology.

The main benefits of our methodology are that it prevents the architect from forgetting some contexts of use, and that it prevents the architect from taking into account too much variability and therefore from wasting time to implement useless ACs. This is because our methodology relies on making explicit the variability of the processes and the different contexts of use of the ACs. Indeed, at Sodifrance we have identified 384 different Java development processes [12]. It is of course impossible for a human being to have in mind all these different processes, and moreover to have in mind all the different contexts of use of all the different ACs.

A limitation of our methodology is that its efficiency depends on the SPML that is used to define the SPrL. Indeed, this SPML may not enable to capture some kinds of information (*e.g.,* a tool definition) and therefore neither does the SPrL. In this case, the architect cannot rely on the SPrL to identify all the contexts of use of the ACs. Only the knowledge of the architect can enable the identification of the contexts of use of the ACs that are not captured by the SPrL. Relying on the knowledge of the architect to identify the contexts of use of the ACs is less efficient than relying on the SPrL. Indeed, the architect can forget some contexts of use or can consider useless contexts of use.

A disadvantage of our methodology is that it is more difficult to identify the contexts of use of the ACs using an SPrL than when the processes are defined in extension (*i.e.,* the common parts between the processes are not factorized). Indeed, it is difficult to visualize in the SPrL what happens before and after a work unit because the SPrL defines the processes in intention instead of extension. This brings difficulties in the identification of the contexts of use of the ACs (*e.g.,* the architect may have difficulties to see if an AC has already been applied and does not need to be applied again).

VI. Related Work

Some approaches provide mechanisms for specifying the variability of a product line (*e.g.,* feature models [13], orthogonal variability models [7]). SPLE [7] relies on this specification of variability in order to produce reusable development artifacts. Several approaches enable to implement reusable artifacts, like object-oriented programming [14], aspect oriented programming [15] or component-based software engineering [16]. But none of these approaches helps to identify the level of reuse of the development artifacts.

Some approaches [1]–[3] address this limitation by providing guidelines to identify the level of reuse of the development artifacts that realize software products, according to the specification of the variability of these software products. More precisely, a feature model is used to specify the variability of a software product. Examples of guidelines are that the hierarchy of the components that realize the software product largely corresponds to the hierarchy of the feature model [1], that alternative features may implement a common interface [1], [2], or that categories of objects can be deduced from categories of features [3]. If we try to apply these approaches to the case of ACs that realize software processes, then the feature model would specify the variability of a family of software processes and the ACs would be the software components for which we want to identify the level of reuse. However, it would not be possible to reuse the guidelines as they are since each feature of the feature model (that corresponds to a fragment of software process) would not be systematically automated by an AC. Hence the necessity of identifying the process fragments that each AC automates, as proposed in our methodology. Another method enables to identify reusable parts of a component by identifying common parts between the variants of a component's variation points [17]. But this method requires the preliminary identification of the variation points and variants of a component, which our methodology helps.

In the field of software processes, there are several approaches that rely on SPrLE [4] to improve the reuse of software processes [8]–[11]. We go further by also using SPrLs to improve the reuse of ACs.

VII. Conclusion and Perspectives

We propose a methodology that improves the identification of the level of reuse of the ACs. It consists of identifying the contexts of use (captured by processes) of the ACs, and in thinking about how to implement the ACs in order to reuse them across their contexts of use.

The benefits of our methodology are that it prevents the architect from forgetting some contexts of use and from taking into account useless contexts of use. The disadvantage of our methodology is that identifying the contexts of use of the ACs is more difficult with an SPrL than when the processes are defined in extension. The limitation is that the efficiency of our methodology depends on the SPML that is used to define the SPrL.

As perspectives of work, we are implementing a tool that supports our methodology and we are applying our methodology on industrial processes of Sodifrance.

References

[1] C. K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "FORM: A feature-oriented reuse method with domain-specific reference architectures," *Annals of Software Engineering*, vol. 5, pp. 143–168, 1998.

[2] K. Lee and K. Kang, "Feature Dependency Analysis for Product Line Component Design," in *ICSR*, 2004, pp. 69–85.

[3] K. Lee, K. C. Kang, W. Chae, and B. W. Choi, "Featured-based approach to object-oriented engineering of applications for reuse," *Softw. Pract. Exper.*, vol. 30, no. 9, pp. 1025–1046, 2000.

[4] H. D. Rombach, "Integrated Software Process and Product Lines," in *ISPW*, 2005, pp. 83–90.

[5] OMG, "Software and Systems Process Engineering Metamodel Specification (SPEM) Version 2," 2008.

[6] ——, "Unified Modeling Language Specifications Version 2," 2005.

[7] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[8] T. Ternité, "Process Lines: A Product Line Approach Designed for Process Model Development," in *SEAA*, 2009, pp. 173–180.

[9] T. Martínez-Ruiz, F. García, and M. Piattini, "Towards a SPEM v2.0 Extension to Define Process Lines Variability Mechanisms," in *SERA*, 2008, pp. 115–130.

[10] J. Hurtado Alegría, M. Bastarrica, A. Quispe, and S. Ochoa, "An MDE Approach to Software Process Tailoring," in *ICSSP*, 2011, pp. 43–52.

[11] J. Alegria and M. Bastarrica, "Building Software Process Lines with CASPER," in *ICSSP*, 2012, pp. 170–179.

[12] E. Rouillé, B. Combemale, O. Barais, D. Touzet, and J.-M. Jézéquel, "Leveraging CVL to Manage Variability in Software Process Lines," in *APSEC*, 2012, pp. 148–157.

[13] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Carnegie-Mellon University Soft. Eng. Institute, Tech. Rep., 1990.

[14] B. Cox, *Object Oriented Programming*. Addison-Wesley,Reading, MA, 1985.

[15] S. Clarke and E. Baniassad, *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, 2005.

[16] I. Crnkovic, *Building Reliable Component-Based Software Systems*. Artech House, Inc., 2002.

[17] W. Zhongjie, X. Xiaofei, and Z. Dechen, "A Component Optimization Design Method Based on Variation Point Decomposition," in *SERA*, 2005, pp. 399–406.